Computing H-Partitions in ASP and Datalog

Chloé Capon, Nicolas Lecomte and Jef Wijsen

University of Mons, Mons, Belgium

Abstract

An H-partition of a finite undirected simple graph G is a labeling of G's vertices such that the constraints expressed by the model graph H are satisfied. These constraints concern the adjacency or nonadjacency of vertices depending on their label. For every model graph H, it can be decided in non-deterministic polynomial time whether a given input graph G admits an H-partition. Moreover, it has been shown in [1] that for most model graphs, this decision problem is in deterministic polynomial time. In this paper, we show that these polynomial-time algorithms for finding H-partitions can be expressed in Datalog with stratified negation. Moreover, using the answer set solver Clingo [2, 3], we have conducted experiments to compare straightforward guess-and-check programs with Datalog programs. Our experiments indicate that in Clingo, guess-and-check programs run faster than their equivalent Datalog programs.

Keywords

Answer Set Programming, Datalog, logic programming, graph partitioning

1. Introduction

Answer Set Programming (ASP) is a powerful programming paradigm that allows for an easy encoding of decision problems in NP. If the answer to a problem in NP is "yes," then, by definition, there is a "yes"-certificate of polynomial size that can be checked in polynomial time. In an ASP *guess-and-check program*, a programmer first declares the format of such a certificate, and then specifies the constraints that a well-formatted certificate should obey in order to be a "yes"-certificate. For example, for the well-known problem SAT, an ASP-programmer can first declare that certificates take the form of truth assignments, and then specify that "yes"-certificates are those certificates that leave no clause unsatisfied.

While ASP guess-and-check programs are typically oriented towards NP-complete problems, they can also be used for problems in P. For example, the previously mentioned encoding of SAT also solves 2SAT, which is known to be in P. In general, assume that we have an answer set solver at our disposal, and that we have written a guess-and-check ASP program for a particular problem that is NP-complete (for example, SAT). Assume furthermore that we know that under some restrictions, the problem can be solved in polynomial time (for example, the restriction of SAT to 2SAT). In logic programming, it may be possible to encode such a polynomial-time solution in a syntactic restriction of ASP, for example, in Datalog with stratified negation [4], which guarantees the existence of a polynomial-time execution. From a theoretical standpoint, a program in Datalog is more efficient than a general guess-and-check ASP program. From a practical standpoint, however, one may wonder whether such a Datalog program will run faster in our ASP engine, compared to a general guess-and-check ASP program. The latter question will be addressed in this paper.

ASPOCP 2022: 15th Workshop on Answer Set Programming and Other Computing Paradigms, July 31, 2022, Haifa, Israel

jef.wijsen@umons.ac.be (J. Wijsen)

http://math.umons.ac.be/staff/Capon.Chloe/ (C. Capon); https://math.umons.ac.be/staff/Lecomte.Nicolas/ (N. Lecomte); http://informatique.umons.ac.be/staff/Wijsen.Jef/ (J. Wijsen)

0000-0001-8216-273X (J. Wijsen)

CEUR Workshop Proceedings (CEUR-WS.org)

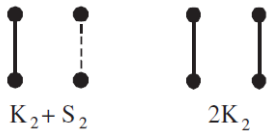


Figure 1: Examples of model graphs (figure copied from [1]).

We will test this scenario on the problem of finding H-partitions in undirected graphs G [1]. All undirected graphs in this paper are understood to be finite and simple. Given an undirected graph G, this problem consists in deciding whether there exists a partition of G's vertices such that this partition respects some constraints encoded in another graph H with exactly four vertices, called *model graph*. Each class of the partition is labeled by a vertex in H. There are two types of constraints:

- if H contains a *full edge* between two vertices, A and B, then each vertex in the class labeled by A must be adjacent to each vertex in the class labeled by B; and
- if H contains a *dotted edge* between two vertices, A and B, then each vertex in the class labeled by A must be nonadjacent to each vertex in the class labeled by B.

Note that there are no constraints between two vertices in the same class. All possible model graphs, up to isomorphism, are presented in Figures 1 and 2. The H-partitioning problem for $K_2 + S_2$, also known as finding a skew partition, is in polynomial time [5, 6]. The H-partitioning problem for $2K_2$ is NP-complete [7]. For each model graph in Figure 2, Dantas et al. [1] provide a polynomial-time algorithm, of low polynomial degree, for deciding whether an undirected graph admits an H-partition. In this paper, we show how these polynomial-time algorithms can be encoded in Datalog with stratified negation. We then experimentally compare these Datalog programs with a guess-and-check ASP program. In theory, the computational complexity of Datalog is in P, while guess-and-check ASP programs can solve NP-complete problems.

This paper is organized as follows. Section 2 formalizes the problem of finding H-partitions and paraphrases the polynomial-time method of Dantas et al. [1]. Sections 3 and 4 focus on finding H-partitions in logic programming, first in ASP, and then in Datalog with stratified negation. In our experimental validation of Section 5, we first generate undirected graphs of different sizes on which our programs can be executed. We distinguish between those input graphs that admit an H-partition, called yes-instances, and those that do not, called no-instances. We show different methods for generating yes-instances and no-instances, for a fixed model graph H, and then compare the running times of our logic programs in the answer set solver Clingo [2, 3]. Finally, Section 6 concludes the paper. Due to space limitations, some proofs have been omitted, which can be found in the extended version [8].

2. Preliminaries

In this section, we formally define the problem H-PARTITION and sketch an algorithm borrowed from [1]. In Section 4, this algorithm will be written in Datalog with stratified negation.

2.1. The Problem H-PARTITION

A *model graph* H is an undirected graph with four vertices, called A, B, C and D (sometimes written in lowercase letters). The vertices are called A, B, C, and D in anticlockwise order, starting with the top-left vertex. Every edge is of exactly one of two types: *full* or *dotted*. All possible model graphs, up to isomorphisms, are presented in Figures 1 and 2. We define H-PARTITION as the following decision problem.

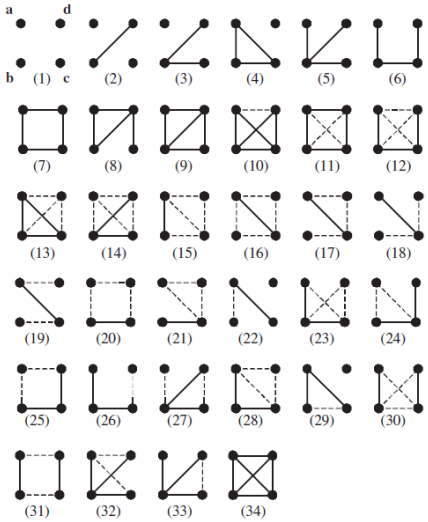


Figure 2: List of model graphs H (figure copied from [1]).

Problem H-PARTITION

Input: A model graph H; an undirected graph G.

Question: Is it possible to partition $V(G)$ into four pairwise disjoint, non-empty subsets, V_A, V_B, V_C, and V_D such that, for all $p, q \in \{A, B, C, D\}$ with $p \neq q$:

- if (p, q) is a full edge of H, then every vertex in V_p is adjacent to every vertex in V_q; and
- if (p, q) is a dotted edge of H, then every vertex in V_p is nonadjacent to every vertex in V_q.

Such a partition, if it exists, is called a *solution* or an *H-partition*.

We call a pair (H, G) a *yes-instance* if H-PARTITION returns *yes* on input H and G; otherwise (H, G) is a *no-instance*. H-PARTITION(H) denotes the H-PARTITION problem for a fixed model graph H.

The partitions V_A, V_B, V_C, and V_D are commonly denoted by A, B, C, and D, respectively. One can view such a partition of $V(G)$ as a labeling of $V(G)$ with the labels A, B, C, and D. From here on, we will often omit curly braces and commas in the denotation of subsets of $\{A, B, C, D\}$. For example, ABD denotes the set $\{A, B, D\}$.

Definition 1. This definition is relative to a fixed model graph H. Let $L \subseteq \{A, B, C, D\}$. We define $\mathsf{N_F}(L)$ as the set of vertices of H that are adjacent, through a full edge, to some vertex of L. Similarly, $\mathsf{N_D}(L)$ is the set of vertices of H that are adjacent, through a dotted edge, to some vertex of L. We say that L is *trivial* if $|L| = 1$.

Informally, given a model graph H, each label among A, B, C, and D imposes a constraint that can be read from H. For example, for the model graph $H = (19)$ given in Figure 2, the label A imposes *"being adjacent to C and nonadjacent to D,"* and C imposes *"being adjacent to A and nonadjacent to B."* A list of two or more labels imposes all the constraints of each label in the list. For example, for $H = (19)$, the list AC imposes *"being adjacent to C, nonadjacent to D, adjacent to A, and nonadjacent to B."* A list is called *conflicting* if it imposes an unsatisfiable constraint. For example, for $H = (20)$, the list AC is conflicting, because A imposes *"being nonadjacent to B,"* while C imposes *"being adjacent to B."* We will call a list *non-maximal*[1] if it can be extended into a longer list that imposes the same constraint. For

[1] In [1], non-maximal lists are called *impossible*.

example, for $H = (33)$, the lists CD and ACD impose the same constraint, namely *"being adjacent to B, nonadjacent to C, and nonadjacent to D,"* and therefore CD is non-maximal.

Definition 2. This definition is relative to a fixed model graph H. Let $L \subseteq \{A, B, C, D\}$. L is *conflicting* if $\mathsf{N_F}(L) \cap \mathsf{N_D}(L) \neq \emptyset$. L is *non-maximal* if there exists $L' \subseteq \{A, B, C, D\}$ such that $L \subsetneq L'$ with $\mathsf{N_F}(L) = \mathsf{N_F}(L')$ and $\mathsf{N_D}(L) = \mathsf{N_D}(L')$.

For example, let H be model graph (13) of Figure 2. The list AB is non-maximal because for the set ABC, we have that $\mathsf{N_F}(AB) = ABC = \mathsf{N_F}(ABC)$ and $\mathsf{N_D}(AB) = D = \mathsf{N_D}(ABC)$. The list AD is conflicting because $B \in \mathsf{N_D}(AD) \cap \mathsf{N_F}(AD)$.

2.2. Finding H-Partitions

Dantas et al. [1] have shown that H-PARTITION(H) is in polynomial time for all model graphs H in Figure 2. Our algorithmic approach slightly differs from theirs because, unlike [1], we will not use non-maximal or conflicting lists. We now describe our approach.

Definition 3. Let H be a model graph, and G an undirected graph. Let x_A, x_B, x_C, and x_D be four distinct vertices of G. We say that the quadruplet (x_A, x_B, x_C, x_D) is *H-isomorphic* if the subgraph of G induced by these vertices is a yes-instance of H-PARTITION(H) for a labeling where x_A, x_B, x_C, and x_D are respectively labeled by A, B, C, and D.

For instance, in the graph of Figure 3a, the quadruplet $(2, 3, 4, 5)$ is H-isomorphic to the model graph (7) of Figure 2.

Our algorithmic approach loops over all H-isomorphic quadruplets (x_A, x_B, x_C, x_D). Each such quadruplet yields an initial partial labeling. Given such an initial partial labeling, we test whether it can be extended into a complete labeling, i.e., into a solution. To this end, we repeatedly pick an unlabeled vertex, and compute its possible labels. When we say that a label $P \in \{A, B, C, D\}$ is *possible* for a vertex, we mean that the model graph does not forbid us to label that vertex with P, given the vertices that have already been labeled. If only one label is possible for an unlabeled vertex, we label that vertex with that label. If no label is possible for some unlabeled vertex, we conclude that the current labeling cannot be extended to a complete labeling. If for every unlabeled vertex at least two labels remain possible, then we conclude that G is a yes-instance for H-PARTITION(H) (except for model graphs (7), (10) and (11) where some additional tests are needed).

In [1], *refined lists* are obtained by removing conflicting and non-maximal lists. The description on [DdFGK05, page 141] might suggest that we must be careful to only consider refined lists. However, the following lemma implies that conflicting or non-maximal lists will not occur in our algorithmic approach.

Lemma 1. *Let H be a model graph, and G an undirected graph. Assume that, during the execution of the previously described algorithm for* H-PARTITION(H) *with input G, there is a vertex $v \in V(G)$ whose set of all possible labels is L with $|L| \geqslant 2$. Then, L is neither conflicting nor non-maximal.*

In Section 4, we show how to encode our algorithmic approach in Datalog with stratified negation. However, we will first provide a straightforward guess-and-check program for H-PARTITION.

3. Guess-and-Check Program for H-PARTITION

In all logic programs of this paper, the constants are in lowercase and the variables are in uppercase to match with Clingo syntax. We use the binary predicate `e` to store the edges of the input graph G,

while the unary predicate `vertex` stores vertices. The binary predicates `full` and `dotted` encode, respectively, full and dotted edges of the model graph. The labels are provided to the program as `partition(a)`, `partition(b)`, `partition(c)`, and `partition(d)`.

The following program follows the *generate-and-test* (or *guess-and-check*) methodology that is classical for problems in NP: generate a candidate solution, and test whether it is a true solution, i.e., whether it satisfies all constraints. In the following program, the generating part is the rule `1 { placedIn(X,P) : partition(P) } 1 :- vertex(X).`, which places every vertex in exactly one partition. The testing part imposes that every partition must contain at least one vertex, and that the constraints of the model graph must be satisfied. Moreover, we suppose that `e` is symmetric, i.e., `e(t,s)` is stored whenever `e(s,t)` is stored.

```
% Every vertex goes in exactly one partition.
1 { placedIn(X,P) : partition(P) } 1 :- vertex(X).

% No partition is empty.
filled(P) :- placedIn(X,P).
:- partition(P), not filled(P).

% Constraints of the model graph.
:- placedIn(X,P), placedIn(Y,Q), full(P,Q), not e(X,Y).
:- placedIn(X,P), placedIn(Y,Q), dotted(P,Q), e(X,Y).
```

4. Datalog Programs for H-PARTITION

In this section, we show how to encode H-PARTITION(H) in Datalog with stratified negation, for model graphs H in Figure 2. Section 4.1 encodes the approach of [1] in general. We argue that this general encoding is correct for every model graph in Figure 2 that is distinct from (7), (10), and (11). We then show how to adjust the general encoding for these three model graphs. Finally, we show that for model graphs H with an isolated vertex, non-recursive Datalog with negation suffices to check the existence of H-partitions.

4.1. General Datalog Program

Dantas et al. [1] showed that H-PARTITION(H) can be solved in polynomial time for all model graphs H in Figure 2. Following our previously described method, we check whether some initial valid labeling of four vertices, called a *base*, can be extended to a complete labeling of all vertices.

Finding a base. The algorithm loops over all quadruplets of distinct vertices and checks whether a quadruplet is H-isomorphic. Our Datalog rules are as follows:

```
base(X,Y,Z,T) :- distinct(X,Y,Z,T), not problematic_base(X,Y,Z,T).
```

Here, the predicate `distinct(X,Y,Z,T)` checks whether X, Y, Z, and T are four distinct vertices of G, and `problematic_base` is defined, for each possible edge in the model graph H, with two rules. For example, if we consider the edge (a, b), the rules are:

```
problematic_base(X,Y,Z,T) :- dotted(a,b), e(X,Y), vertex(Z), vertex(T).
problematic_base(X,Y,Z,T) :- full(a,b), not e(X,Y), vertex(X), vertex(Y), vertex(Z), vertex(T).
```

Similar rules are added for edges (a, c), (a, d), (b, c), (b, d), and (c, d).

Extension to a complete labeling. First, for every base (I, J, K, L) of four vertices, we label I with A, J with B, K with C, and L with D.

```
inPart(I,a,I,J,K,L) :- base(I,J,K,L).
inPart(J,b,I,J,K,L) :- base(I,J,K,L).
inPart(K,c,I,J,K,L) :- base(I,J,K,L).
inPart(L,d,I,J,K,L) :- base(I,J,K,L).
```

We keep track that a vertex that has been labeled in a base must not be re-labeled later on.

```
done(I,I,J,K,L) :- base(I,J,K,L).
done(J,I,J,K,L) :- base(I,J,K,L).
done(K,I,J,K,L) :- base(I,J,K,L).
done(L,I,J,K,L) :- base(I,J,K,L).
```

Next, rather than computing the set of possible labels for a yet unlabeled vertex X, we use the predicate `imp(X,P,I,J,K,L)` with the meaning that the vertex X cannot be labeled by $P \in \{A, B, C, D\}$, relative to the fixed base (I, J, K, L). The Datalog rules are straightforward. The second rule, for example, expresses that for a full edge (P, Q) in the model graph, if a vertex Y has been labeled with Q, then a vertex X that is nonadjacent to Y cannot be labeled with P. One should read "prob" as a shorthand for "problematic."

```
imp(X,P,I,J,K,L) :- vertex(X), full_prob(X,P,I,J,K,L), not done(X,I,J,K,L).
full_prob(X,P,I,J,K,L) :- full(P,Q), inPart(Y,Q,I,J,K,L), not e(X,Y), vertex(X).

imp(X,P,I,J,K,L) :- vertex(X), dot_prob(X,P,I,J,K,L), not done(X,I,J,K,L).
dot_prob(X,P,I,J,K,L) :- dotted(P,Q), inPart(Y,Q,I,J,K,L), e(X,Y).
```

For every vertex X of G that is not in the fixed base (I, J, K, L), we now consider two possibilities:

(1) if three labels among A, B, C, and D are impossible for X, then X is labeled with the remaining fourth label; and

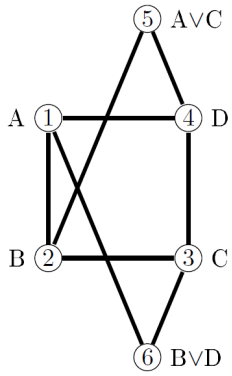
(2) if all labels among A, B, C, and D are impossible for X, then the base cannot be extended to a complete labeling.

Finally, if some base never encounters the second case, the answer to H-PARTITION(H) is "yes"; otherwise the answer is "no". Note incidentally that in a "yes"-instance, there may be vertices X that are not labeled by the first case.

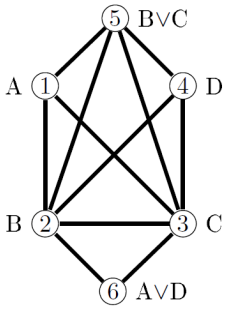
```
% First case.
inPart(X,a,I,J,K,L) :- imp(X,b,I,J,K,L), imp(X,c,I,J,K,L), imp(X,d,I,J,K,L).
inPart(X,b,I,J,K,L) :- imp(X,a,I,J,K,L), imp(X,c,I,J,K,L), imp(X,d,I,J,K,L).
inPart(X,c,I,J,K,L) :- imp(X,a,I,J,K,L), imp(X,b,I,J,K,L), imp(X,d,I,J,K,L).
inPart(X,d,I,J,K,L) :- imp(X,a,I,J,K,L), imp(X,b,I,J,K,L), imp(X,c,I,J,K,L).

% Second case.
bad_init(I,J,K,L) :- vertex(X), base(I,J,K,L), imp(X,a,I,J,K,L),
                     imp(X,b,I,J,K,L), imp(X,c,I,J,K,L), imp(X,d,I,J,K,L).

yes_instance() :- base(I,J,K,L), not bad_init(I,J,K,L).
no_instance() :- not yes_instance().
```

We have the following result.

Theorem 2. *If H is one of the model graphs of Figure 2 such that H is distinct from model graphs (7), (10), and (11), then* H-PARTITION(H) *is expressible in Datalog with stratified negation.*



(a) Counterexample for the model graph (7).



(b) Counterexample for the model graph (10).

Figure 3: The corner cases of the main approach.

4.2. Adjustment for Model Graph (7)

In [1], the model graph (7) of Figure 2 is reduced to the model graph K_2. This is possible because the constraints for A and C are identical (i.e., *being adjacent to both B and D*), and the constraints for B and D are identical. Then, every non-base vertex labeled by C can also be labeled by A, and every non-base vertex labeled by D can also be labeled by B. Our previous Datalog program would fail, however, for the reason that the set of possible labels for a non-base vertex is never a singleton.

For example, consider the graph G in Figure 3a with the base $(1, 2, 3, 4)$. The list of possible labels for the vertex 5 is AC, and the list of possible labels for the vertex 6 is BD. Our previous algorithm would terminate here and conclude that G is a yes-instance. Note, however, that if vertex 5 is labeled with A (the case where we label 5 with C is similar), it is impossible to label the vertex 6 with B or D, because vertices 5 and 6 are nonadjacent.

We adjust our Datalog program as follows: if the set of possible labels of a vertex X has size ≥ 2, the new program will label X with one of these two possibilities, arbitrarily chosen. The rest of the program remains unchanged. That is, in the box that immediately precedes Theorem 2, we replace the four rules for the predicate `inPart` by the following two rules:

```
inPart(X,a,I,J,K,L) :- imp(X,b,I,J,K,L), imp(X,d,I,J,K,L).
inPart(X,b,I,J,K,L) :- imp(X,a,I,J,K,L), imp(X,c,I,J,K,L).
```

4.3. Adjustment for Model Graphs (10) and (11)

As in [1], the model graphs (10) and (11) of Figure 2 need a special treatment. We illustrate the issue by the model graph (10) and the graph G given in Figure 3b. For the base $(1, 2, 3, 4)$, the list of possible labels for the vertex 5 is BC, and the list of possible labels for the vertex 6 is AD. So our original algorithm would terminate and wrongly conclude that the base $(1, 2, 3, 4)$ can be extended to a complete labeling. Nevertheless, we claim that this base cannot be extended to a complete labeling. Indeed, let us label the vertex 5 by B (the case where we label 5 with C is similar). Since B is adjacent, through full edges, to both A and D, and since vertices 5 and 6 are nonadjacent, we cannot label 6 with A or D. We note incidentally that the graph G is a yes-instance through other bases, for example, $(1, 2, 3, 6)$.

It can be verified that for the model graph (10), the lists of possible labels are AD and BC; for the model graph (11), the lists of possible labels are AC and BD. We have the following result.

Lemma 3. *(Cf. [1]) Let H be one of the model graphs (10) or (11) in Figure 2. If there exists a solution to*

H-PARTITION(H) *with a base* (i, j, k, l)*, then there is also a solution with the same base such that:*

- *each vertex having* A *in its list of possible labels is labeled* A*; and*
- *each vertex having* B *in its list of possible labels is labeled* B*.*

We now use Lemma 3 to adjust our previous Datalog program of Section 4.1. First, we copy the labeled vertices.

```
label(X,a,I,J,K,L) :- inPart(X,a,I,J,K,L).
label(X,b,I,J,K,L) :- inPart(X,b,I,J,K,L).
label(X,c,I,J,K,L) :- inPart(X,c,I,J,K,L).
label(X,d,I,J,K,L) :- inPart(X,d,I,J,K,L).
```

Next, if B is an impossible label for a vertex but A is not, the vertex is labeled by A. Symmetrically, if A is an impossible label for a vertex but B is not, the vertex is labeled by B.

```
label(X,a,I,J,K,L) :- not imp(X,a,I,J,K,L), imp(X,b,I,J,K,L).
label(X,b,I,J,K,L) :- imp(X,a,I,J,K,L), not imp(X,b,I,J,K,L).
```

We now have to check that this labeling is correct according to the model graph. If not, we reject the initial base.

```
problem(X,P,I,J,K,L) :- label(X,P,I,J,K,L), full(P,Q), label(Y,Q,I,J,K,L), not e(X,Y).
problem(X,P,I,J,K,L) :- label(X,P,I,J,K,L), dotted(P,Q), label(Y,Q,I,J,K,L), e(X,Y).
bad_base(I,J,K,L) :- problem(X,P,I,J,K,L).
```

Finally, the definition of `yes_instance()` is changed as follows:

```
yes_instance() :- base(I,J,K,L), not bad_base(I,J,K,L), not bad_init(I,J,K,L).
no_instance() :- not yes_instance().
```

Along the lines of Theorem 2, we obtain the following result.

Theorem 4. *If* H *is one of the model graphs* (10) *or* (11) *in Figure 2, then* H-PARTITION(H) *is expressible in Datalog with stratified negation.*

4.4. Model Graphs with Isolated Vertices

A vertex in a model graph H is *isolated* if it is not adjacent to a full or dotted edge. The model graphs in Figure 2 with at least one isolated vertex are (1), (2), (3), (4), (22), and (29). It can be easily seen that for these model graphs, H-PARTITION(H) can be solved in non-recursive Datalog with negation (i.e., in predicate logic).

Lemma 5. *Let* G *be an undirected graph. Let* H *be a model graph with an isolated vertex. Then the following are equivalent:*

1. G *contains an* H*-isomorphic quadruplet of distinct vertices; and*
2. G *is a yes-instance of the problem* H-PARTITION(H)*.*

From Section 4.1, it follows that non-recursive Datalog suffices to test the existence of an H-isomorphic quadruplet of distinct vertices. Thus, we obtain the following result.

Theorem 6. *If* H *is a model graph with an isolated vertex, then* H-PARTITION(H) *is expressible in non-recursive Datalog with negation.*

5. Experiments

To compare the efficiency of our programs in an existing answer set solver, we need a generator for yes-instances and no-instances of arbitrary size. The motivation for distinguishing between yes- and no-instances comes from the asymmetry of NP-problems: on a yes-instance, an algorithm can stop as soon as the first H-partition is found; but on no-instances no such early stopping is possible. In this section, we discuss generators for yes- and no-instances, and then we report on execution times of our programs on these instances.

5.1. Generating Yes-Instances

Let n, m be two positive integers, and H a model graph. We want to build a graph G with n vertices and m edges that is a yes-instance for the problem H-PARTITION(H). Necessarily, $n \geqslant 4$ and $m \leq \frac{n(n-1)}{2}$, where the latter is the number of edges in the complete graph with n vertices.

We denote $\mathbb{N}_0 = \{1, 2, 3, \ldots\}$. Let $a, b, c, d \in \mathbb{N}_0$ such that $a + b + c + d = n$. One can wonder if there exists a yes-instance with m edges where the numbers of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d.

Definition 4. We define $m^H_{min}(a, b, c, d)$ as the smallest number k such that some graph G with $|E(G)| = k$ admits an H-partition (V_A, V_B, V_C, V_D) such that $|V_A| = a$, $|V_B| = b$, $|V_C| = c$, and $|V_D| = d$. Analogously, we define $m^H_{max}(a, b, c, d)$ as the largest number k such that some graph G with $|E(G)| = k$ admits an H-partition (V_A, V_B, V_C, V_D) such that $|V_A| = a$, $|V_B| = b$, $|V_C| = c$, and $|V_D| = d$.

We write p for a label and also for the number of vertices labeled with this label. If the context is not clear, we write $\#p$ for the number of vertices labeled by p. We have the following result.

Lemma 7. *Let H be a model graph. Let $a, b, c, d \in \mathbb{N}_0$ such that $a + b + c + d = n$. Then,*

$$m^H_{min}(a, b, c, d) = \sum_{(p,q) \in \mathsf{Full}(H)} \#p \cdot \#q \tag{1}$$

$$m^H_{max}(a, b, c, d) = \frac{n(n-1)}{2} - \sum_{(p,q) \in \mathsf{Dotted}(H)} \#p \cdot \#q \tag{2}$$

where $\mathsf{Full}(H)$ *is the set of the full edges of H, and* $\mathsf{Dotted}(H)$ *is the set of the dotted edges of H.*

Proof. We give a proof of (1). We first show that there exists an undirected graph G_{min} with n vertices and $\sum_{(p,q) \in \mathsf{Full}(H)} \#p \cdot \#q$ edges that has a solution where the numbers of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d. Let G_{min} be an undirected graph whose vertices are

$$V(G_{min}) = \{A_1, A_2, \ldots, A_a, B_1, \ldots, B_b, C_1, \ldots, C_c, D_1, \ldots, D_d\}$$

and the edges of G_{min} are defined as follows: for $p_i, q_j \in V(G_{min})$, (p_i, q_j) is an edge of G_{min} if and only if (p, q) is a full edge of H.

Since for each full edge (p, q) in H, each vertex labeled with p is connected to each vertex labeled with q, there are $\#p \cdot \#q$ such edges. Moreover, there are no other edges than those described here. As we sum up these values for each full edge (p, q), we obtain the desired number of edges in G_{min}.

We claim that this labeling satisfies all the constraints. The full constraints are satisfied by construction. The dotted constraints are also satisfied because G_{min} contains only edges connecting two vertices whose labels are adjacent through a full edge in H.

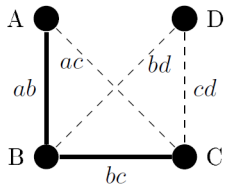


Figure 4: Model graph (23).

It remains to be shown that there exists no graph with strictly less than $\sum_{(p,q)\in \mathsf{Full}(H)} \#p \cdot \#q$ edges that is a yes-instance where the number of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d. Assume for the sake of contradiction that there exists such a graph G'. We can assume without loss of generality that G' and G_{min} have the same set of vertices. Since G' has strictly less edges than G_{min}, there is an edge (u, v) in G_{min} that is not in G'. Since G' is a yes-instance, it has a solution. Let p be the label of u in this solution, and q the label of v. Since (u, v) is an edge of G_{min}, we have that (p, q) is a full edge of H. So, we have two vertices u and v that are labeled by p and q respectively, but there is no edge in G' between u and v. This is a contradiction with the fact that G' is a yes-instance. This concludes the proof of (1).

The proof of (2) is similar because it suffices to consider G_{max} which is obtained from a complete graph with n edges by removing all the edges between two vertices whose labels are connected by a dotted edge in H. □

For example, let us fix $a, b, c, d \in \mathbb{N}_0$ with the model graph (23) of Figure 2 and $n = a + b + c + d$. According to the Figure 4, we have:

$$m^{(23)}_{min}(a, b, c, d) = ab + bc$$

$$m^{(23)}_{max}(a, b, c, d) = \frac{n(n-1)}{2} - ac - bd - cd$$

Moreover, we claim that, for all m such that $m^{(23)}_{min}(a, b, c, d) \leqslant m \leqslant m^{(23)}_{max}(a, b, c, d)$, there exists a graph G with n vertices and m edges such that G is a yes-instance for a labeling where the numbers of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d. The idea is that we can extend the graph for $m^{(23)}_{min}(a, b, c, d)$ by adding edges until we reach m edges such that every added edge does not violate any dotted constraint. We will now formalize this construction.

Definition 5. Let G and G' be two undirected graphs, We say that G is an *extension* of G', denoted $G' \subseteq G$, if $V(G') = V(G)$ and $E(G') \subseteq E(G)$. If $G' \subseteq G$, we also say that G' is *extended by* G.

In other words, G is an extension of G' if it is possible to obtain G from G' by adding some edges. One can easily see that $G_{min} \subseteq G_{max}$. Indeed, we take G_{min} and consider the labeling induced by the construction of G_{min}. We can add all possibles edges that do not violate any dotted constraint. In this way, we obtain G_{max} by definition.

Theorem 8. *Let H be a model graph. Let $a, b, c, d \in \mathbb{N}_0$ and $n = a + b + c + d$. Let G_{min} and G_{max} be the graphs with n vertices and respectively $m^H_{min}(a, b, c, d)$ and $m^H_{max}(a, b, c, d)$ edges defined in (the proof of) Lemma 7. Every graph G such that $G_{min} \subseteq G \subseteq G_{max}$ has a solution (V_A, V_B, V_C, V_D) where $|V_A| = a$, $|V_B| = b$, $|V_C| = c$, and $|V_D| = d$.*

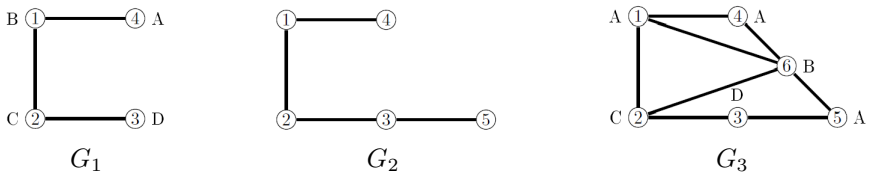


Figure 5: For the model graph (6), we have that G_1 is a yes-instance, G_2 is a no-instance, and G_3 is a yes-instance.

Proof. Let us consider the same labeling of the vertices in G as in G_{min}. So the number of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d. It remains to be shown that this labeling satisfies all the constraints.

- Let (p, q) be a full edge of H. We have to show that all the vertices labeled by p are adjacent to all vertices labeled by q. This holds true because G is an extension of G_{min} and G_{min} already satisfies this constraint.
- Let (p, q) be a dotted edge of H. We have to show that all the vertices labeled by p are nonadjacent to all vertices labeled by q. This holds true because G_{max} is an extension of G and G_{max} already satisfies this constraint.

All the constraints are satisfied and, consequently, G is a yes-instance. □

This theorem has a natural corollary.

Corollary 9. *Let H be a model graph, and $a, b, c, d \in \mathbb{N}_0$ such that $n = a + b + c + d$. Let m be an integer such that $m^H_{min}(a, b, c, d) \leqslant m \leqslant m^H_{max}(a, b, c, d)$. There is an undirected graph G with n vertices and m edges such that G has a solution where the numbers of vertices labeled by A, B, C, and D are, respectively, a, b, c, and d.*

Based on Corollary 9, we introduce the following generator of yes-instances.

Generator for yes-instances

Input: A model graph H; two positive integers n and m such that $n \geqslant 4$.

Output: Yes-instances with n vertices and m edges.

Generator: For every quadruplet (a, b, c, d) of positive integers such that $a + b + c + d = n$ and $m^H_{min}(a, b, c, d) \leqslant m \leqslant m^H_{max}(a, b, c, d)$,

> build a yes-instance from G_{min} by randomly adding edges that do not violate any dotted constraint, until the number of edges reaches m;

if no such quadruplet exists, return *"there is no such yes-instance."*

5.2. Generating No-Instances

Given an integer $n \geqslant 4$ and a model graph H, we want to build a graph G with n vertices that is a no-instance for the problem H-PARTITION(H). The following proposition tells us when this is possible.

Proposition 10. *Let H be a model graph and n an integer such that $n \geqslant 4$. There is an undirected graph G with n vertices that is a no-instance for* H-PARTITION(H) *if and only if H is not the model graph (1) in Figure 2.*

Proof. Assume that H is the model graph (1). Then, any graph G with n vertices is a yes-instance because there is no constraint to satisfy, due to the absence of full and dotted edges.

For the opposite direction, assume that H is not the model graph (1). Then H contains a dotted edge or a full edge.

- If H contains at least one full edge, the empty graph with n vertices and no edge is a no-instance because the full constraint will never be satisfied for any labeling of the vertices.
- If H contains at least one dotted edge, the complete graph with n vertices and $\frac{n(n-1)}{2}$ edges is a no-instance because the dotted constraint will never be satisfied for any labeling of the vertices.

This concludes the proof. □

The undirected graphs in the proof of Proposition 10 are of little interest in an experimental validation, because they do not admit an H-isomorphic quadruplet of vertices, and therefore the Datalog program would not even enter its recursive part. Instead, for a model graph H, we would like to generate no-instances that contain an H-isomorphic quadruplet. Note here that, by Lemma 5, such no-instances do not exist if H has an isolated vertex.

It would be interesting to develop an approach similar to that for yes-instances: start with a small no-instance and extend it until some desired number of vertices is reached. Nevertheless, this seems a difficult task, because the addition of edges for new vertices can turn no-instances into yes-instances, and vice versa. For example, in Figure 5, G_1 is extended by G_2, and G_2 is extended by G_3. For the model graph (6), we have that G_2 is a no-instance, while G_1 and G_3 are yes-instances.

In view of the aforementioned difficulties, we have decided to generate no-instances in a randomized fashion. Given $n \geqslant 4$ and model graph H that is not the model graph (1) of Figure 2, the idea is to generate random graphs with n vertices until a no-instance for H-PARTITION(H) is found. To see whether this approach is viable, we have generated random graphs with $n = 100$ by adding an edge to a graph with a given probability, its expected density. The expected density is considered as a random variable whose probability distribution is a Gaussian distribution, $\mathcal{N}(0.5, 0.25)$. This distribution seems better than a "classical" uniform distribution on the expected density because there are more possible graphs with a density of 0.5 than 0.01 or 0.99. Table 1 shows, for each model graph in Figure 2, the number of yes-instances and no-instances over a sample of 1000 generated graphs.

For the model graphs (1), (2), (3), (4), (18), (22), (29) of Figure 2, there are more yes-instances than no-instances. Except for (18), these model graphs have an isolated vertex and, by Lemma 5, an input graph is a yes-instance as soon as it contains an H-isomorphic quadruplet. For the model graph (18), a graph is easily a yes-instance because it suffices to have an edge (u, v) such that u is nonadjacent to at least two other vertices: the vertex u will be labeled by C, all vertices adjacent to u will have the label A, and all nonadjacent vertices will have label B or D. For model graphs that are distinct from the above seven model graphs, there are considerably more no-instances than yes-instances. In conclusion, our approach should find a no-instance in a few tries, except for the seven models that have been discussed before, for which we will use the no-instances of the proof of Proposition 10.

5.3. Execution Times

We have used the answer set solver Clingo [2, 3] to compare the execution times of ASP guess-and-check programs with programs in Datalog with stratified negation. Figure 6 shows execution times on input graphs of different small sizes that were generated as previously explained. The execution times reported

model graph	# yes	# no	model graph	# yes	# no
(1)	1000	0	(18)	999	1
(2)	1000	0	(19)	185	815
(3)	1000	0	(20)	15	985
(4)	998	2	(21)	35	965
(5)	15	985	(22)	1000	0
(6)	175	825	(23)	15	985
(7)	13	987	(24)	15	985
(8)	15	985	(25)	14	986
(9)	13	987	(26)	56	944
(10)	9	991	(27)	15	985
(11)	0	1000	(28)	5	995
(12)	0	1000	(29)	1000	0
(13)	0	1000	(30)	0	1000
(14)	2	998	(31)	0	1000
(15)	4	996	(32)	6	994
(16)	4	996	(33)	15	985
(17)	183	817	(34)	12	988

Table 1
Proportion of yes- and no-instances.

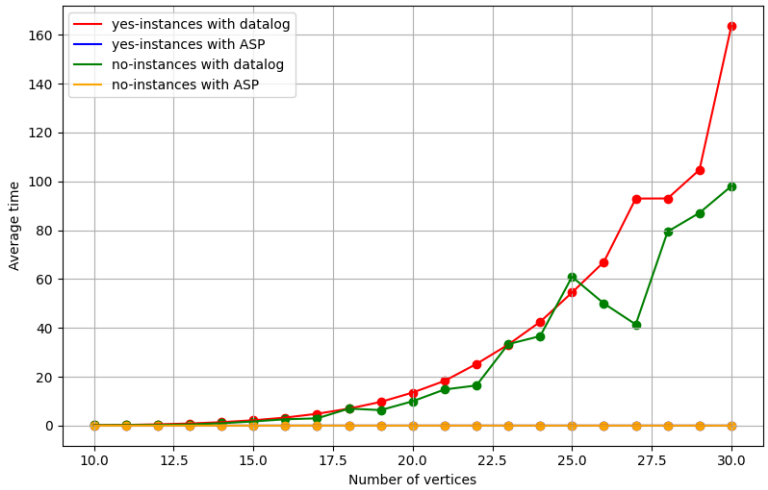


Figure 6: Execution time in Clingo (the blue line is hidden behind the yellow line).

in Figure 6 are average execution times over the 34 model graphs of Figure 2. It is immediately clear from this figure that when executed by Clingo, the guess-and-check program is much more efficient than our Datalog programs with stratified negation. The difference in efficiency was even more striking for larger graphs. In fact, on graphs with more than 50 vertices, we encountered memory problems when executing our Datalog programs in Clingo, which seem to be due to the size of the grounded program. In search for an explanation, we displayed and investigated the grounded rules in Clingo. The grounded Datalog programs only contain true atoms, i.e., rules whose body has already been evaluated as true and therefore left empty. Figure 7 shows that the Datalog programs yield significantly more grounded rules than the guess-and-check programs. Datalog often yields more than 250 000 true atoms while guess-and-check never results in more than 5000 ground rules.

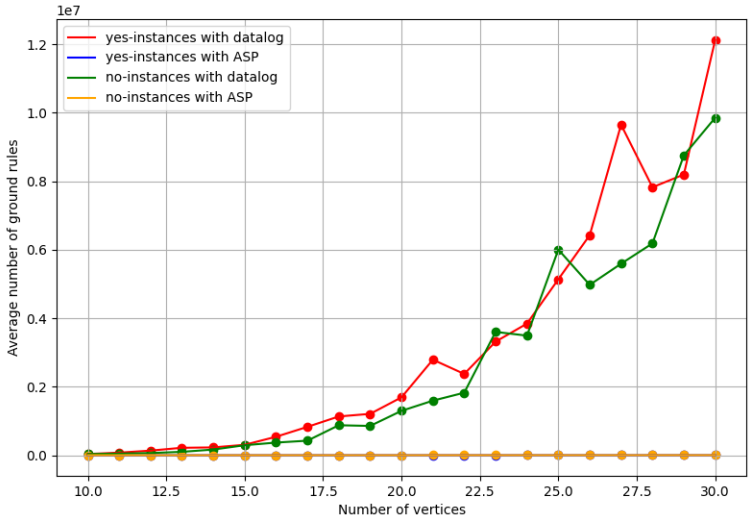


Figure 7: Number of true atoms (Datalog) and grounded rules (ASP) in Clingo (the blue line is hidden behind the yellow line).

6. Conclusion

We presented a logical programming approach to the problem of finding H-partitions in undirected graphs. In particular, we showed how the polynomial-time solutions given in [1] can be encoded in Datalog with stratified negation. In an experimental validation, we found that in the answer set solver Clingo, these Datalog programs execute much slower than a simple guess-and-check program. We also studied the problem of generating yes- and no-instances for H-PARTITION with a fixed number of vertices or edges.

Our experiments indicate that when using an answer set solver, there may be no benefit in writing programs that are theoretically more efficient than guess-and-check programs. It would be interesting to investigate execution times of our Datalog programs on a dedicated Datalog engine. It would also be interesting to find systematic methods for generating no-instances with a fixed number of vertices or edges.

References

[1] S. Dantas, C. M. H. de Figueiredo, S. Gravier, S. Klein, Finding H-partitions efficiently, RAIRO Theor. Informatics Appl. 39 (2005) 133–144. URL: https://doi.org/10.1051/ita:2005008. doi:10.1051/ita:2005008.

[2] M. Gebser, R. Kaminski, A. König, T. Schaub, Advances in *gringo* series 3, in: J. P. Delgrande, W. Faber (Eds.), Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings, volume 6645 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 345–351. URL: https://doi.org/10.1007/978-3-642-20895-9_39. doi:10.1007/978-3-642-20895-9_39.

[3] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo = ASP + control: Preliminary report, CoRR abs/1405.3694 (2014). URL: http://arxiv.org/abs/1405.3694. arXiv:1405.3694.

[4] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

[5] C. M. H. de Figueiredo, S. Klein, Y. Kohayakawa, B. A. Reed, Finding skew partitions efficiently, J. Algorithms 37 (2000) 505–521. URL: https://doi.org/10.1006/jagm.1999.1122. doi:10.1006/jagm.1999.1122.

[6] W. S. Kennedy, B. A. Reed, Fast skew partition recognition, in: H. Ito, M. Kano, N. Katoh, Y. Uno (Eds.), Computational Geometry and Graph Theory - International Conference, KyotoCGGT 2007, Kyoto, Japan, June 11-15, 2007. Revised Selected Papers, volume 4535 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 101–107. URL: https://doi.org/10.1007/978-3-540-89550-3_11. doi:10.1007/978-3-540-89550-3_11.

[7] C. N. Campos, S. Dantas, L. Faria, S. Gravier, $2K_2$-partition problem, Electron. Notes Discret. Math. 22 (2005) 217–221. URL: https://doi.org/10.1016/j.endm.2005.06.031. doi:10.1016/j.endm.2005.06.031.

[8] C. Capon, N. Lecomte, J. Wijsen, Computing h-partitions in ASP and datalog, CoRR abs/2202.03730 (2022). URL: https://arxiv.org/abs/2202.03730. arXiv:2202.03730.